*URL:* [*http://www.elsevier.nl/locate/entcs/volume55.html*](http://www.elsevier.nl/locate/entcs/volume55.html) *12 pages*

*Model Checking with Abstract Types Kirsten Winter 1*

*Software Veri cation Research Center SVRC, University of Queensland, Queensland 4072 Australia*

*Abstract*

*Model checking the design of a software system can be supported by providing an interface from a high-level modelling language, which is suitable for describing software design, to a given model checking tool. In order to cope with the higher complexity of software systems, we additionally need a means for reducing the sys- tem's state space. This can be done be applying abstraction to large or in nite data parts of the model. In our work, we introduce an interface from the high-level mod- elling language ASM to Multiway Decision Graphs (MDGs). Similar to OBDDs, MDGs are a data structure suitable for symbolic representation of transition sys- tems, and their model checking. Since MDGs support the representation of abstract sorts and functions they can treat abstract models. We present a transformation algorithm from ASM to MDGs that automatically generates abstract models once the user has marked the data to be abstracted. We adapt the MDG model checking algorithms for the treatment of ASM models.*

# *1 Introduction*

*Most model checkers operate on models that are given in a low-level language* that is developed for specifying hardware circuits rather than software. The development of software, however, should be supported by a high-level lan- guage that provides usual language facilities such as complex data types, and appropriate structuring mechanisms. When using model checking to support analysing software models, we have to provide a transformation from such a high-level language to the input language of a chosen model checker. Such a transformation has to bridge the gap between the di erent languages in a se- mantic preserving way that, moreover, avoids adding complexity to the model checking task as far as possible.

*In general, software systems involve more complexity than hardware sys-* tems due to their more complex data part. One means to cope with this

*1 Email:* [*kirsten@svrc.uq.edu.au*](mailto:kirsten@svrc.uq.edu.au)

*c 2001 Published by Elsevier Science B. V. Open access under* [*CC BY-NC-ND license.*](http://creativecommons.org/licenses/by-nc-nd/3.0/)

*complexity is abstraction: We may reduce the state space of a model by ap-* plying an abstraction function that provides a lifting of in nite sets into nite sets of equivalence classes that are relevant for the behaviour of the model. Two additional tasks arise within this approach: An appropriate abstraction function has to be de ned and the abstract model that preserves the properties to be checked has to be computed.

*In our work, we use Multiway Decision Graphs (MDGs) (see* [CZS+97,CCL+97]) as a basic data structure for implicit state enumeration and model checking ([XCS+98]). Similar to the OBDD approach that is used for SMV ([McM93]) and other tools, MDGs are suitable for representing state transition systems and provide eÆcient algorithms for computing and check- ing their reachable states. In contrast to OBDDs, however, MDGs are able to represent values of any enumeration set (rather than being restricted to boolean values) and support the use of abstract sorts and functions. This provides a means of avoiding expensive boolean encoding (as in OBDD-based approaches) and to treat possibly in nite systems.

*In [XCS+98], MDGs are used for model checking circuit designs given* in a hardware description language, called MDG-HDL. In our approach, we aim at using the MDG approach for supporting a more general speci cation language, called Abstract State Machines (ASM, [Gur95]). ASM has been used in academic and industry contexts to develop systems in a variety of domains (a bibliography is given in [Hug]). ASM model transition systems in a simple and uniform fashion and provide an operational semantics. A transformation from ASM into the input language of a transition system-based model checker appears feasible as rst results with the SMV tool show (see [Win97,CW00]). By representing ASM models by means of MDGs, we bene t from a straightforward transformation (similar to the approach used for interfacing the SMV tool) and gain a very simple means for generating abstract models: We extend the ASM language with a notion of abstract types. If the user chooses an in nite or large data type to be abstract then consequently all functions that are applied to this type turn into abstract functions that are left uninterpreted during the state exploration. Predicates over abstract data automatically provide a suitable partitioning into equivalence classes. This di erent treatment of abstract functions and predicates is provided automat-

*ically during our transformation step from ASM into MDGs.*

*Due to the corresponding paradigm of both notations and the aim of eÆ-* cient representation, we do not map ASM onto the given hardware description language MDG-HDL but rather provide a mapping from ASM to the MDG data structure itself. Consequently, we are not supported with a black box model checking tool that is ready to use but rather with a library of all neces- sary functions for computing MDGs. That is, we rebuild a model checker for the needs of ASM based on this library according to the work that is done for model checking MDG-HDL models ([Xu99,XCS+98]). Apart from the map- ping of ASM to MDGs, we therefore have to adapt some algorithms that are

*developed for model checking circuits.*

*An outline of the paper follows: Section 2 and Section 3 introduce ASM* and MDGs, respectively. In Section 4, we show our transformation from ASM into the graph structure of MDGs. Our model checking approach for ASM using MDGs is introduced in Section 5. We conclude this work with related and future work in Sections 6 and 7.

# *2 Abstract State Machines*

*Abstract State Machines (ASM) is a high-level language suitable for a wide* range of domains (for a bibliography see [Hug]). It is a state-based approach for describing transition systems: The states are given in terms of sorts (do- mains) and functions over these sorts. Functions can be static and have a constant value, dynamic and thus be controlled by the system during a run (similar to state variables), or external and controlled by the environment (like input variables). The behaviour of the system is given as a set of transition rules which speci es the updating of the dynamic functions depending on the current state of the system. The ASM notation provides a skip rule, a simple update rule that changes the value of (i.e., updates) a dynamic function, a block rule that gathers a set of rules, a conditional rule that restricts a rule to re only in states that satisfy the guarding condition (which is a rst- order predicate), a do-forall rule that applies a parameterised rule for all members of a given set, and a choose rule that introduces non-determinism by applying a parameterised rule for one arbitrarily chosen element of a given set 2 . During a run of the system, all transition rules re simultaneously, i.e., all rules are applied in a single step and lead to the next state of the system. For a full de nition of ASM see ,e.g., [Gur95].

*We give a short impression of the language by means of a simple example,*

*a generic timer (see Fig. 1):* The system gets a natural number max as an input that speci es the number the timer has to count to. The timer has two states COUNT, which is the initial state, and RING. As long as it is in state COUNT it increments the value of an

if (t<max) then incr(t)

*Fig. 1: Example of a generic timer*

max reset(t)

if (t=max)

COUNT

RING

reset(t)

*internal variable t. If t reaches max, the system changes to state RING, indi-*

*cating that the time is elapsed. It then resets t and changes back into the* initial state COUNT.

*When modelling the generic timer in ASM, we introduce two sorts,*

*2 ASM supports also an import rule that allows the user to introduce fresh elements. However, this extension of sorts cannot be treated by our model checking approach.*

*MODE = fcount; ringg and TIME = N , and two dynamic functions over these, namely mode:MODE and t:TIME. Initially, mode equals count and t equals 0. To model the input, we introduce an external function max:TIME. Two static functions model incrementing and reseting of the timer variable, incr = ft 7! (t + 1) j t 2 TIMEg and reset = ft 7! 0 j t 2 TIMEg. The ASM transition rules are now given as follows:*

*if (mode = count) ^ (t < max) then t := incr(t)*

*if (mode = ring)*

*then mode := count t := reset (t)*

*if (mode = count ) ^ (t = max) then mode := ring*

*A run of this ASM model is de ned as a sequence of states starting with* the initial state that is de ned through the initial values of dynamic func- tions. In every state, the next state is derived by applying the transition rules simultaneously to the current state.

*Abstract ASM. In order to exploit the support of abstract data types* provided by MDGs, we introduce a syntactic feature to label any sort as being an abstract sort. In the example given above, we might change the sort TIME to be abstract.

*Functions over abstract sorts do not have a xed interpretation. They allow* for any interpretation that matches their signature. Abstracting from sorts is a means of lifting a \concrete" ASM model into an \abstract" ASM model whose instances comprise concrete models for all possible interpretations of the abstract sorts and functions.

abstract model

*model checking MDG representation*

*gabs : Dataabs Dataabs ! Dataabs*

*fabs : Dataabs ! Bool*

*stripping off interpretation:*

*Q : Dataabs*

*append : Q Q ! Q is empty : Q ! Bool*

*mult : Int Int ! Int*

*is even : Int ! Bool*

*op : W ord W ord ! W ord is int : W ord ! Bool*

concrete model

other concrete models as instances

*Fig. 2. Lifting a concrete model to an abstract model*

*Figure 2 depicts the abstraction step: We consider the sort Q in our con-* crete model as abstract and change all its occurrences into Dataabs. As a result, we get an abstract model of the same signature. For this abstract speci cation, all those interpretations are suitable that have a sort, a 2-ary function that maps arguments of the given sort to a value of the same sort, and a boolean predicate over the sort. In the gure, we give some examples for di erent interpretations for the sort Dataabs and the functions that are possible.

*The purpose of this abstraction step is to substitute in nite sorts, and func-* tions over them, since these cannot be exhaustively explored. When checking an abstract model, abstract functions (i.e., functions that map into an abstract sort) are left uninterpreted. Functions over some abstract sorts that map to a concrete nite sort (as, for instance, boolean predicates over abstract pa- rameters) can be investigated by means of a complete case distinction. This naturally provides a partitioning of the in nite sort into nitely many equiv- alence classes. The state space of the abstract model is thus smaller in most cases. It can be canonically represented by MDGs as we show in the following sections.

# *3 Multiway Decision Graphs*

*Multiway Decision Graphs (MDGs) are a generalisation of Binary Decision* Diagrams. They are a data structure for canonically representing formulas of a many-sorted rst-order logic, called Directed Formulas (DFs). A special feature of the underlying logic is the distinction between concrete and abstract sorts. Correspondingly, function symbols may be concrete, abstract (if the range is abstract), or cross-operators (if the range is concrete but the domain contains some abstract sort).

*DFs are suitable to describe sets of states and transition relations of tran-* sition systems. They are formulas in disjunctive normal form (DNF) over simple equations of the following form: f (B1;::: Bn) = a (where f is a cross- operator and a is a constant of concrete sort), w = a (where w is a variable of concrete sort and a is a concrete constant), or v = A (where v is a variable of abstract sort and A is a term of the same sort). Furthermore, in each disjunct of a DF, all left hand sides (LHSs) of the equations are pairwise distinct and every abstract variable that occurs as a LHS must occur in every disjunct of the DF. To be represented by MDGs, a DF has to be concretely reduced, i.e., all concrete variables that occur on the right-hand side (RHS) of an equation have to be substituted by a value.

*An MDG is a nite graph G, whose non-terminal nodes are labelled by* terms and whose edges, starting from a non-terminal node, are labelled by terms of the same sort as the node. Terminal nodes are labelled by formulas. Generally, a graph G represents a formula in the following way:

*If G consists of a single terminal node, then it represents the formula the*

*node is labelled with.*

*If G has a root node labelled with term A and edges labelled with terms B1;::: ; Bn leading to subgraphs G1;::: ; Gn that represent formulas P1;::: ; Pn, then G represents the formula*

*(A = Bi ^ P1) \_ (A = B2 ^ P2) \_ ::: \_ (A = Bn ^ Pn)*

*In Figure 3, we depict the* formula that is given above as a graph G. To be a canonical repre- sentation, an MDG has to satisfy certain well-formedness conditions which involve an order on function

*A*

*B1*

*B*

*n*

*B2*

...

*symbols and variables that has G1 G2* to be provided by the user (the

*G*

*n*

*detailed list of conditions is de ned* in [CZS+97,CCL+97]).

*Fig. 3: The MDG G*

*A library of operations on MDGs is available that is suÆcient for realising* an implicit state enumeration, namely disjunction, relational product, and pruning-by-subsumption. They are de ned for combining sets rather than only pairs of MDGs. This allows us to represent the transition relation as a set of several small graphs instead of one big graph and bene ts from a more eÆcient state enumeration.

*The relational product operation computes the conjunction of a set of* graphs under existential quanti cation of all variables in a given variable set

*E and the possible renaming of variables, ((9v 2 E)(V1 i n Pi) ). This*

*is used for computing the set of reachable states from a given state (in which*

*case the MDGs Pi represent the transition relation and E the set of state* variables). Pruning-by-subsumption approximates the di erence of two sets. This enables us to check if an invariant (given as an MDG) is satis ed in a given set of states (also given as an MDG).

*According to the well-formedness conditions on MDGs, the application of* the operations is restricted. The relational product of two MDGs can only be computed if their nodes are not labelled with the same abstract variable. Disjunction, in contrast, is applicable only to MDGs that contain the same set of abstract variables as nodes labels (i.e., the same set of abstract variables that occur as LHSs in the equations of the corresponding DF).

# *4 The Transformation from ASM to MDGs*

*The transformation between the two notations is split into two steps. By in-* troducing an intermediate language, called ASM-IL, we bene t from a general interface that can be reused for connecting other model checkers or state- transition-based tools. Due to the limitation of space, we give only a short overview of the transformation algorithm. A more detailed description can be

*found in [Win01].*

*ASM into ASM-IL. Due to the fact that any set of ASM transition* rules can be attened into a set of simple guarded update rules, we can fully represent ASM models { except models that contain import-rules 3 { as a set of location-update pairs. A location is a dynamic function applied to a value, which can change its value over a run (similar to a state variable). A location-update pair is a tuple of a location and its possible guarded updates, (loc; f(guard j ; val j) j 0 j ng). We gather this information for each

*location loci of the ASM-model. The set of all location-update pairs represents*

*the model in ASM-IL.*

*Transforming ASM into ASM-IL involves two steps: (a) unfolding dy-* namic functions into locations if they occur as LHSs of updates (e.g., f (x) with x 2 fa; bg results in f a and f b) or substituting them by their pos- sible values otherwise; (b) attening all (nested) transition rules into simple guarded update rules over only one location. This way, we produce a lot of code. However, predicates can be resolved and simpli ed due to the partial evaluation in the rst step.

*If this procedure meets an abstract function unfolding is avoided, the func-* tion is left uninterpreted. If it meets a function of concrete sort that is applied to abstract terms it treats this term as a cross-term (i.e., a cross-operator applied to some abstract terms), and avoids unfolding as well. If the cross- operator matches one of the standard relational operators (e.g., =, , etc.) the tool automatically introduces a new symbol that can be distinguished (e.g., isEq, leq, etc.). For instance, the predicate (a b), where a and/or b are abstract terms, is mapped into leq(a; b). These new cross-operators auto- matically introduce a partitioning of the abstract sort suitable for the model.

*ASM-IL as MDGs. It can be shown that each location-update pair* in ASM-IL can be represented as a well-formed MDG: Due to the fact that during the rst step of our transformation we unfold every concrete function that is not on the LHS of an update, location-update pairs are \concretely reduced" (in MDG-terminology). Moreover, each pair contains at most one abstract variable that appears on the LHS of an equation or update, namely loc. Any other equation in a guard that has an abstract variable as its LHS is transformed into a cross-term. We can deduce that every case in the location- update pair contains the same LHS abstract variable, namely loc or none.

*According to the de nition of MDGs, each location-update pair* (loci; f(guard ij ; val ij) j 0 j ng) can be represented in the graph structure

*as follows (where loc0 denotes the value of loci in the next state): The*

*i*

*location loc0*

*i*

*labels the root node of the graph. Each edge starting at the*

*root is labelled with one of the speci ed update values val ij and leads to* the subgraph Gij that represents the corresponding guard of the update

*3 Choose rules are simulated by introducing an external variable for making the non- deterministic choice.*

*guardij. Figure 4 sketches a graph for a location-update pair containing* three possible guarded updates.

*Since in ASM the state remains* unchanged if none of the guards is satis ed we have to add an explicit else-case to the MDG (otherwise the next value is arbitrary). The subgraph Gelse represents the formula

*:(guard i1 \_ ::: \_ guard in). In case loci is of concrete type, we have to*

*vali1*

*Gi1*

*loc0*

*vali2 vali3*

*i*

*Gi2 Gi3*

*loci*

*Gelse*

*substitute the edge-label with a path* for each possible value of loci.

*Fig. 4: Location-update pair as MDG*

*The transformation of guards is based on the fact that simple equations* eq(lhs; rhs) assemble a new MDG where the root is labelled by lhs and the only edge, labelled with rhs, leads to the leaf true. Conjunction and disjunction can be computed by the corresponding operations on MDGs once the operands are represented as MDGs. Negation is only possible if the operand does not contain abstract node labels.

# *5 Adapted Model Checking Algorithm*

*[XCS+98,Xu99] introduce a rst-order temporal logic, called LMDG, and cor-* responding model checking algorithms based on MDGs. LMDG is the universal fragment (i.e., it excludes the existential temporal operator) of Abstract CTL\*, a deviation of CTL\* that has an additional construct for quanti cation of vari- ables.

*With the usual meaning of temporal operators, a formula in LMDG may have the form A', AG', AF', A'U , AG(' ) (F )), or AG(' ) ( 1U 2)), where '; ; 1; and 2 are so called Next let formulas. The Next let formulas are de ned as follows (where model variables are the input-, state- and output-variables of the system model we want to check):*

*The boolean truth values T, F, and any equation t1 = t2 are* Next let formulas (where t1 is a model variable and t2 is a model variable, an ordinary variable, a function over those, or a constant).

*If p and q are Next let formulas then so are: :p, p ^ q, p \_ q, p ! q, Xp and LET (v = t) IN p (where t is a model variable and v is an ordinary variable).*

*LMDG is restricted to simple templates of formulas with temporal path* operators (A, AG, AF, etc.) as given above and does not support nest- ing of those. Furthermore, the nesting of X in Next let formulas is lim- ited. These two restrictions enable simple checking algorithms for this lan- guage ([Xu99,XCS+98]): The Next let formulas occurring in a formula to be checked are represented as a circuit consisting of and-gates, or-gates, compara-

*tors and registers. Its output is a ag that indicates the truth-value of the* Next let formula. This circuit is composed with the system model M such that M outputs those model-variables to the circuit the Next let formula refers to. That is, the circuit determines the truth value depending on some particular state variables of the system. The overall model checking algorithms (each formula template is treated by a particular algorithm) compute the reachable states of the composed machine and check the truth-value of the ag (if it equals true the property is satis ed).

*For model checking ASM models represented as MDGs, we can use the* same model checking algorithms since they are based on the MDG library. However, we have to change the representation of Next let formulas. They must be transformed into simple ASM models consisting of some additional variables (and their initialisation) and simple guarded transitions rules which can be transformed into ASM-IL.

*Any Next let formula p can be represented as an ASM Mp. The composed* model of system model M and Mp is then given as M extended by the addi- tional variables and the transition rules of Mp. We build Mp in the following way: We introduce a new dynamic function Flagp : Bool which represents the truth value of p and

*if p is of the form T, F or (t1 = t2) then it is represented by an ASM Mp* with the following transition rule and initialisation of the ag variable:

*if p then Flagp := true else false initially Flagp = p;*

*if o and q are Next let formulas represented by Flago and Flagq then the Next let formula p with*

*p = :q is represented by*

*if :Flag q then Flagp := true else false initially Flagp = :Flagq ;*

*p = o ^ q is represented by*

*if (Flago ^ Flagq ) then Flagp := true else false initially Flagp = (Flago ^ Flagq );*

*p = o \_ q is represented by*

*if (Flago \_ Flagq ) then Flagp := true else false initially Flagp = (Flago \_ Flagq );*

*p = o ! q is simpli ed to p = :o \_ q.*

*p = Xq is represented by*

*if Flagq then Flagp := true else false initially Flagp = true;*

*p = LET (v = t) IN q is represented by the following (where we introduce*

*a new variable v of the same sort as t)*

*if ((v = t) ^ Flagq ) then Flagp := true else false initially Flagp = ((v = t) ^ Flagq );*

*In each of the given templates for Mp, the location Flagp represents the*

*truth value of formula p. Note that for any formula p = Xq, the initialisation* of Flagp equals true, i.e., Flagp equals the truth value of p in the next state only. For model checking AGp, AFp, etc. we apply the composed machine of M and Mp to the corresponding checking algorithm for AG, AF, etc. These algorithms basically compute the reachable states and check if Flagp = true holds always or eventually, etc. depending on the algorithm (for more detail see [Xu99,XCS+98]).

# *6 Related Work*

*Uninterpreted functions are addressed elsewhere: In [BD94] data values and* operations within the speci cation of the DLX architecture are modelled by means of uninterpreted functions. However, this approach allows only va- lidity checking, no temporal properties can be checked. [CN94] introduce a new logic, called GTL, which also allows uninterpreted functions to be rep- resented. The decidable fragment of GTL can be treated by an automatic validity checker (based on PVS). The logic LMDG, however, is more expres- sive then the decidable fragment of GTL (as proven in [Xu99]) and model checking algorithms go beyond validity checking.

*Due to the support for abstract sorts in MDGs, the computation of the* abstract model appears to be much simpler than the mechanisms suggested in, e.g., [GS97] and [BPR00]. Instead of providing an abstraction function and proving that properties are preserved, we generate with less e ort an abstract model that includes the intended model and more. This may result in false negatives, that is counter-examples that are not related to the particular instance of the model we want to check (in this case, it may be possible to add rewriting rules to exclude the non-intended interpretations), but if no counter-example can be found then all instances are correct.

*Some process algebras such as CRL ([BP95]), or XMC ([RRS+00]) allow* the user to specify complex data types abstractly. These languages are also supported by model checking. They are well suited to model communicating distributed system. However, modelling a state-based software system by means of a process algebra can be quite diÆcult.

# *7 Conclusion*

*In this work, we presented a transformation from the high-level modelling* language ASM ([Gur95]) into the data structure MDG ([CZS+97,CCL+97]). This transformation enables us to represent ASM models as MDGs and exploit their library of functions which are suitable for implementing model check- ing algorithms. We showed how the suggested approach for model checking LMDG([Xu99,XCS+98]), a sub-language for rst-order CTL\*, can be adapted to the ASM language. The bene t of this approach is a more eÆcient coding of ASM: Instead of forcing a binary encoding of complex data types (as within

*OBDD-based tools), MDGs support a more compact representation of multi-* valued data types (less state variables have to be introduced). Moreover, we gain simple support for generating abstract models by means of abstracting large or in nite data types. MDGs allow for representation of abstract values and functions, of which the latter are left uninterpreted.

*This approach appears to be promising for analysing software systems since* complex data parts can be easily abstracted. Relational operators provide a naturally given partitioning of data types according to their in uence in the control ow (e.g., whenever data are requested in the guards of a transition rule). The abstract model and the partitioning of abstract types is provided automatically by the transformation step.

*The implementation of the transformation is completed. The next step is* to code the model checking algorithm and show the feasibility of the approach.

# *Acknowledgement*

*I would like to thank Graeme Smith and also the anonymous reviewers for* useful comments on earlier versions of this paper.

# *References*

*[BD94] J. Burch and D. Dill. Automatic veri cation of pipelined microprocessor control. In D.L. Dill, editor, Proc. of Int. Conf. on Computer Aided Veri cation, CAV'94, volume 818 of LNCS, pages 68{80. Springer-Verlag, 1994.*

*[BP95] D. Bosscher and A. Ponse. Translating a process algebra with symbolic data values to linear format. In U.H. Engberg, K.G. Larsen, and*

*A. Skou, editors, Procs. of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'95, BRICS Notes Series, pages 119{130, Aarhus, 1995.*

*[BPR00] T. Ball, A. Podelski, and S.K. Rajamani. Boolean and cartesian abstraction for model checking C programs. Technical Report MSR-TR- 2000-115, Microsoft Research, 2000. To appear in TACAS'2001.*

*[CCL+97] E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, and Z. Zhou. Automated veri cation with abstract state machines using Multiway Decision Graphs. In T. Kropf, editor, Formal Hardware Veri cation: Methods and Systems in Comparison, volume 1287 of LNCS, pages 79{*

*113. Springer Verlag, 1997.*

*[CN94] D Cyrluk and P. Narendran. Ground temporal logic: A logic for hardware veri cation. In D. Dill, editor, Proc. of Int. Conf. on Computer Aided Veri cation, CAV'94, volume 818 of LNCS, pages 247{259. Springer- Verlag, 1994.*

*[CW00] G. Del Castillo and K. Winter. Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, Proc. of 6th Int. Conference for Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000, volume 1785 of LNCS, pages 331{346. Springer-Verlag, 2000.*

*[CZS+97] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway Decision Graphs for automated hardware veri cation. Formal Methods in System Design, 10(1), 1997.*

*[GS97] S. Graf and H. Sa di. Construction of abstract state graphs with PVS. In*

*O. Grumberg, editor, Proc. of Int. Conf. on Computer Aided Veri cation, CAV'97, volume 1254 of LNCS, pages 72{83. Springer-Verlag, 1997.*

*[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Borger, editor, Speci cation and Validation Methods. Oxford University Press, 1995.*

*[Hug] J.K. Huggins. Abstract state machines home page. EECS Department, University of Michigan.* [*http://www.eecs.umich.edu/gasm/.*](http://www.eecs.umich.edu/gasm/)

*[McM93] K. McMillan. Symbolic Model Checking. Kluwer Academic Publishers, 1993.*

*[RRS+00] C.R. Ramakrishnan, I.V. Ramakrishnan, S. Smolka, Y. Dong, X. Du,*

*A. Roychoudhury, and V.N. Venkatakrishnan. XMC: A locic- programming-based veri cation toolset. In E.A. Emerson and A.P. Sistla, editors, Proc. of Int. Conf. on Computer Aided Veri cation, CAV 2000, volume 1855 of LNCS, pages 576{579. Springer-Verlag, 2000.*

*[Win97] K. Winter. Model Checking for Abstract State Machines. J.UCS Journal for Universal Computer Science (special issue), 3(5):689{702,* *1997.*

*[Win01] K. Winter. Model Checking Abstract State Machines. PhD thesis, Technical University of Berlin, Germany, 2001. To appear.*

*[XCS+98] Y. Xu, E. Cerny, X. Song, F. Corella, and O. Ait Mohamed. Model checking for a rst-order temporal logic using Multiway Decision Graphs. In Proc. of Int. Conf. on Computer Aided Veri cation (CAV'98), volume 1427 of LNCS, pages 219{231. Springer-Verlag, 1998.*

*[Xu99] Y. Xu. Model Checking for a First-order Temporal Logic Using Multiway Decision Graphs. PhD thesis, University of Montreal, 1999.*